REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	5/16/95	Final Technical Report 8/1/94-3/31/95

4. TITLE AND SUBTITLE

SCALABLE DATA PARALLEL ALGORITHMS AND IMPLEMENTATIONS FOR VISION

5. FUNDING NUMBERS

F49620-94-1-0431

6. AUTHOR(S)

R. Nevatia
V.K. Prasanna

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

University of Southern California
Institute for Robotics & Intelligent Systems
Powell Hall 204
Los Angeles, CA 90089-0273

8. PERFORMING ORGANIZATION REPORT NUMBER

AFOSR-TR- 95-0434

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Advanced Research and Projects Agency
3701 No. Fairfax Drive, Arlington, VA 22203-1711

Air Force Office of Scientific Research
Bldg. 410 , Bolling AFB, DC 20332-6448

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for public release;
Distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

Our research is about designing, analyzing and implementing scalable parallel solutions to problems in intermediate- and high-level vision. This is a difficult problem as computations are heterogeneous, symbolic and geometric in nature and use complex data structures such as lists and graphs. We propose a realistic model of distributed memory parallel machines which accurately models the features of a parallel machine. This includes the costs of communication latency, impact of communication patterns on network congestion, available bandwidth and time for synchronization. We analyze the computation communication and control characteristics and the memory requirements of the vision algorithms. Our parallel algorithms achieve load balancing by dynamic redistribution of the tasks. We show the results of our approach in parallelizing the line finding problem on IBM SP-2 and a perceptual grouping step on TMC CM-5.

14. SUBJECT TERMS

Parallel processing, computer vision, image understanding, line finding, perceptual grouping

15. NUMBER OF PAGES

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	Unlimited

SCALABLE DATA PARALLEL ALGORITHMS

AND

IMPLEMENTATIONS FOR VISION

Grant No. F49620-94-1-0431

Final Technical Report

August 1, 1994 - March 31, 1995

Contractor: University of Southern California

Principal Investigators:

Ramakant Nevatia
Institute for Robotics & Intelligent Systems
School of Engineering
University of Southern California
Los Angeles, CA 90089-0273
(213) 740-6427

Viktor K. Prasanna
Department of Electrical Engineering-Systems
School of Engineering
University of Southern California
Los Angeles, CA 90089-2562
(213) 740-4483

AFOSR Program Manager:

Abraham Waksman
AFOSR
110 Duncan Ave., Suite B115
Bolling AFB, DC 20332-0001

1 Objectives

This effort is about designing, analyzing and implementing scalable parallel solutions to problems in intermediate- and high-level vision. This is a difficult problem as computations are heterogeneous, symbolic and geometric in nature and use complex data structures such as lists and graphs. Simple data parallel approaches are not sufficient due to the need for non-local communication and data dependent load distribution. Such problems require development of a sophisticated computational model and techniques. Specifically, the work has the following area of emphases:

1. Design of scalable parallel algorithms and analysis of their performance for a variety of generic geometrical computational problems, such as perceptual grouping, arising in intermediate and high level vision.

2. Developing a generic, realistic model of computation of parallel machines that includes the local memory, communication latency, bandwidth and synchronization overheads, that spans a variety of MIMD architectures and is usable for a variety of symbolic computations.

3. Test the methodology by implementing an integrated vision system that begins with an image and produces high-level descriptions on a versatile MIMD machine such as a Thinking Machine CM-5 or IBM SP-2. Building detection in aerial images is one of the chosen tasks.

2 Approach

Complexities of the proposed problem preclude the use of massive parallelism based on automatic parallelization or compiler generated mappings. Our approach consists of the following steps:

1. Accurately modeling the features of a parallel machine to develop an abstract coarse grain parallel model of computation that includes the costs of communication latency, impact of communication patterns on network congestion, available bandwidth and time for synchronization.

2. Analyzing the computation, communication and control characteristics and the memory requirements of the vision algorithms.

3. Reorganizing the algorithms to achieve a better match between the characteristics of the computation and the machine. This will typically involve reorganizing the process into computation and communication phases with appropriate synchronization mechanisms so that effects of network communication latency and congestion to not dominate the computation performed by the processors.

4. Achieving load balancing by dynamic redistribution of the tasks.

In the following sections, we show the results of our approach in parallelizing the line finding problem on IBM SP-2 and a perceptual grouping step on TMC CM-5.

3 Status of the Research Effort

We have parallelized the linear feature extraction task, a time consuming procedure in IU. Our implementations have been performed on IBM SP-2. These show linear speed-up compared with serial implementations. We also implement a perceptual grouping task (Line Grouping) on TMC CM-5. In the following, the definition of these IU tasks are given. A realistic model of distributed memory parallel machine is discussed. Keys ideas of the design of fast algorithms for these tasks are discussed. The implementation details and experimental results are shown. These results will appear at the IEEE Workshop on Computer Architectures for Machine Perception, 1995.

3.1 Problem Definition

We first briefly define the selected IU problems to be solved.

A. Linear Feature Extraction

The Linear Feature Extraction consists of *Contour-Pixels Detection* and *Linear Approximation* phases. We describe this algorithm as follows [4] :

The objective of the contour-pixels detection is to extract contour pixels from an input image. The input is a 2-D image array of pixels (grey levels) and the output is a same sized array with directed contour pixels embedded in the array. The contour-pixels detection consists of edge detection, thinning, and linking operations. These are *window operations*, in which the output at a pixel is based on the value of the input pixel and the value of its neighboring pixels. The neighborhood of a pixel is defined by the given window size. Given an $n \times n$ image and fixed number of windows of size $m \times m$, extraction of contour pixels can be performed in $O(m^2n^2)$ time on a serial machine.

Linear approximation is a data reduction technique to extract line segments from a contour-pixel array. The input is a 2-D contour pixel array with directions and the output is a set of line segments approximating the contours. The linear approximation consists of contour tracing and approximation operations. Several heuristics are available for approximating a contour by a set of piecewise line segments [1]. In this paper, we employed a *strip-based* algorithm [8]; given a starting point of a contour and an error bound for controlling the quality of the approximation, it follows the contour pixels until the current pixel is the last pixel of the contour or exceeds a given error bound. A straight line from the starting pixel to the current pixel is used to approximate the contour. If the current pixel exceeds the error bound, then this procedure continues again with this pixel as the starting pixel.

The analysis of our algorithm as well as its implementation can be easily modified to suit other heuristics for approximation to lead to similar performance.

B. Perceptual Grouping

In this section, we briefly describe the processing steps in perceptual grouping.

In general, vision system operates both in bottom-up (feature extraction, grouping) and top-down fashion (verification). The perceptual grouping process is usually an intermediate-level procedure to group the primitive features detected by low-level processing to form structural hypotheses.

We consider parallelizing the perceptual grouping tasks described in [11]. The grouping procedures are performed in a building detection system for grouping linear features to form structural objects such as rectangles. The rectangles are then used to hypothesize building structures. For the sake of completeness, we briefly outline the processing steps in perceptual grouping. Additional details can be found in [11].

(1) **Line Grouping** - groups line segments which are closely bunched, overlapped, and parallel to each other to form a *line* (a linear structure at a higher granularity level). For each line segment, a search is performed within the region on both sides of the line segment within a constant width to find other line segments which are parallel to it. The detected segments are grouped to form a *line*. (2) **Junction Grouping** - groups two close right-angled *lines* to form a *Junction*. For each *line*, a search is performed on both sides of the *line* within a constant width and a fixed size region near its end-points to find *lines* which may jointly form right-angled corner(s). For any two lines which form a "T" junction, the top line will be broken to form two separate lines. To distinguish the linear features detected at different grouping tasks, let *linear* denote the new generated lines and the lines that are not broken. (3) **Parallel Grouping** - groups two *linears* which are parallel to each other and have "high" percentage of overlap. For each *linear*, a search is performed on a window of size $w \times w$ where w is a given value representing the length of the side of a possible building in the image. We then form a *parallel* by grouping the *linear* with the *linear* found in the window having a difference of slope within a given threshold value and satisfying certain constraints with respect to overlap. (4) **U-contour Grouping** - forms a *U-contour* if any *parallel* has its two *linears* aligned at one end. A search is performed within the window near the aligned end of each *parallel* to group with *linears* possibly connecting the end-points at the aligned end. If any two *U-contours* share the same *parallel*, a rectangle is formed as a building hypothesis.

To reduce the search time, we store pointers representing image features in an *index array* of the same size as the image. For example, a pointer stored at (x, y) may point to a *junction* feature formed by two segments with (x, y) as their intersection point. Thus, to find a junction near a point of interest only a neighboring area need to be searched.

Let $W(S)$ denote the total area of all the *search windows* generated by a set S of input *tokens* in a grouping task. Note that the set S represents the token data such as *segment*, *line*, *linear*, and *parallel* in Line, Junction, Parallel, and U-contour Grouping tasks respectively. We store *segment*, *line*, and *linear* tokens in the index array before starting Line Grouping, Junction Grouping, and Parallel Grouping tasks respectively. We assume that a constant number of token data is stored at a grid point of the index array. In fact, the thinning technique used in [11] produces at most three linear features at a grid point. We have $W(S) = \sum_S(c \times l_i)$ for Line Grouping and $W(S) = \sum_S(2m^2 + d \times l_i)$ for Junction Grouping, where l_i is the length of the ith *segment* (*line*), c and d are constants specifying

the width of the search window on each side of the *segment* (*line*), and m^2 denotes the size of the search area at one end of a line. Given a $n \times n$ image, $W(S) = O(n^2)$. Also, $O(n^2)$ time is required to construct the index array. Thus, the Line and Junction Grouping tasks can be performed in $O(n^2)$ time on a sequential machine. The Parallel and U-contour Grouping tasks can be combined into a single task as the search on aligned end overlaps with the search area of Parallel Grouping. The combined task can be completed in $O(|S|w^2 + n^2)$ time on a sequential machine.

3.2 A Model of Distributed Memory Machines

For our analysis, we model the distributed memory machine as a set of high-performance serial machines interacting through a low-latency high-bandwidth network. Interprocessor communication is performed using explicit message passing. We consider moderately parallel processing systems in which the machine sizes are not "large." In general, the size of the machine (the number of processors) is less than a thousand for most commercially available machines such as TMC CM-5, IBM SP-2, and Intel Paragon that have been installed. In the current generation of interconnection networks, the small network (hardware) latency, the high bandwidth of the communication network, and the advent of efficient routing methods, have made the effects of network link contention and the distance between processors (in terms of hops) be relatively small compared with large software overheads in message passing. Traditional task allocation/mapping algorithms consider the interconnection topology and physical distance between the processors. Such approaches have limited capabilities in reducing the the communication time in using state-of-the-art distributed memory machines.

Let P be the number of processors. In our discussion, processing node (PN), processing element (PE), and processor are used interchangeably. Let T_d denote the startup time for sending a message. Let τ_d denote the transmission rate (seconds per unit of data) for data communication. The *startup* time, including the software and communication protocol overheads, is associated with each communication step. We make the following assumptions for our analysis: (1) Sending a message containing m units of data from a processor to another processor or exchanging a message of size m between a pair of processors takes $T_d + m\tau_d$ time. (2) Suppose each processor has m units of data to be routed to a single destination and the set of all destinations is a permutation, then the data can be routed in $T_d + m\tau_d$ time. (3) To perform a *global operation* such as broadcast a data, a barrier synchronization, or a reduction operation (sum, min, max, prefix sum etc.), τ_g time is required. We assume τ_g is a constant, if the machine has a control network dedicated to performing fast global operations, (for example, CM-5 [15]). Otherwise, the operation is implemented using point-to-point communication primitives, $\tau_g = O((\log P)T_d)$.

For most message-passing parallel machines, the ratio of T_d to τ_d is in the range of several hundreds to few thousands as shown in Table 1. To reduce the communication time, the high startup cost, as well as the message length, need to be considered. For communicating many short messages between a pair of processors, the algorithm may combine messages into a larger message block to be transmitted as a single unit between the processors. For communicating long messages, the scheduling of the communication steps to avoid node contention

Machine	T_d (μsec)	$\tau_d(\mu sec/byte)$	T_d/τ_d
CM-5	86	0.12	716
SP-2	46	0.035	1314
Paragon	82	0.26	315
iPSC/860	60	0.50	120
iPSC/2	700	0.36	1944

Table 1: Communication parameters of various message-passing machines.

(several processors attempt to communicate with the same processor simultaneously), and the computation time for preparing the outgoing message need to be taken into account.

3.3 Fast Parallel Algorithms

In this section, we discuss the key ideas of the design of fast algorithm for the selected IU tasks.

A. Parallel Linear Feature Extraction

Let P denote the number of processing nodes. The $n \times n$ image array is divided into P blocks of size $\frac{n}{\sqrt{P}} \times \frac{n}{\sqrt{P}}$. The contour-pixels detection phase can be parallelized by performing the window operations in each processing node. Since the work load for each window operation is nearly the same and each operation can be done independently, we can exploit the data parallelism in each operation naturally. However, some communication is required to exchange boundary data for subsequent window operation. This communication overhead can be reduced by overlapping the computation with the communication.

We assume that the input to the linear approximation phase is the output of the contour-pixels detection phase. The contour-pixels data are stored in a 2-D contour pixel array. In the linear approximation, there are two kinds of data to be handled: *local* contour data and *global* contour data. Local contour is a contour whose pixels are located in a single processing node, while global contour is a contour whose pixels belong to more than one processing node. The local contours can be processed independently as there is no data dependency between the processing nodes. However, in the case of global contours, the processing can start only after the neighboring processing node completes the approximation on the pixels ahead of the local starting pixel.

Following terminology is used in this paper: (1) *starting pixel*: the first pixel of a contour, (2) *ending pixel*: the last pixel of a contour, (3) *local-starting pixel*: the starting pixel of a global contour located in a processing node, (4) *local-ending pixel*: the ending pixel of a global contour located in a processing node, (5) *task*: the computational work to be performed on a local contour or part of a global contour located in a processing node, (6) *token*: data sent from a processing node to another processing node to activate the processing of the next segment of a global contour located in the neighboring processing node. This contains information to continue global contour processing, (7) *ready queue*: queue for tasks having

a token, (8) *wait queue*: queue for tasks not having a token.

A.1 – A Synchronous Iterative Algorithm

A possible solution to linear approximation phase is to use a *synchronous iterative* technique. Such a technique has been widely used in scientific computations [16]. In this approach, each processing node performs operations on its local data, and then checks for a termination condition. If the condition is not satisfied, then all the processing nodes exchange data and proceed to the next iteration. For the purpose of comparison, we outline the synchronous linear approximation algorithm in Figure 1 based on the technique in [7].

Method 1 : A Synchronous Iterative Algorithm
Step 1: Create wait queue.
Step 2: Update the wait queue and ready queue.
Step 3: Take a task from ready queue and perform linear approximation.
Repeat this Step until the ready queue is empty.
Step 4: Synchronize all processing nodes.
Step 5: Check termination condition. If TRUE, terminate.
Step 6: Exchange tokens and go to Step 2.
end

Figure 1: An outline of a Synchronous Iterative Algorithm for linear approximation.

In Step 1, we extract the starting and local-starting pixels from a 2-D contour pixel array and store them into a wait queue. Initially, the contours having the starting pixel are considered as the tasks having a token. Then, we check the wait queue in Step 2. If any task has a token, it is extracted from the wait queue and inserted into the ready queue. All the ready tasks are moved into the ready queue. In Step 3, we take one ready task and perform the contour tracing and the approximation operations. Each processing node has eight outgoing buffers for the eight neighboring nodes. If the task corresponds to a global contour, the token is stored into its corresponding outgoing buffer. After finishing all the ready tasks, the processing nodes participate in the synchronization in Step 4. In Step 5, we combine the status of all the outgoing buffers. If any of the buffers is not empty, communicate with each other in Step 6 and proceed to the next iteration.

In this algorithm, there are two sources of synchronization overheads. First, all processing nodes synchronize (in Step 4) before starting the next iteration. If a processing node completes Step 3 earlier, then it will be idle until all the other processing nodes complete Step 3. This synchronization overhead occurs in practice due to unbalanced work load among processing nodes in an iteration. The second source of synchronization overhead is the time to perform the synchronization command in SP-2. As the number of processing nodes increases, the execution time of the command increases. Thus, the execution time depends on the distribution of the contours among the processing nodes, and in the worst case it can result in poor speed-ups.

The load unbalance problem can be solved with the load re-distribution algorithms [3, 7]. In these methods, the contours are redistributed initially to balance the load on the processing nodes. However, the cost of calculating the load on the processing nodes and load re-distribution overhead make these algorithms attractive only if the computational cost associated processing the with contour data is high.

A.2 – An Asynchronous Algorithm

Method 2 : An Asynchronous Algorithm
Step 1: Count the number of sends and receives.
Step 2: Create wait queue.
Step 3: Receive tokens. Check wait queue.
If no *token* for global contours,
go to Step 5.
Step 4: Take a task for global contour from ready queue and perform the approximation.
If needed, send token. Go to Step 6.
Step 5: Take a task for local contour from ready queue and perform the approximation.
Go to Step 6.
Step 6: Check termination condition.
If FALSE, go to Step 3.
Step 7: Synchronize all processing nodes and terminate.
end

Figure 2: An outline of an Asynchronous Algorithm for linear approximation.

To reduce the overhead in detecting the termination status, we count the number of outgoing and incoming global contours in each processing node. Thus, the termination status in each processing node is reached when all the queues become empty and the processing node finishes sending the tokens. The termination status of the overall system can be detected using a single collective communication command. This elimination of synchronization overhead can improve the utilization of processing nodes as each processing node can start its next task independently of all the other processing nodes.

In order to improve the execution time further, we intentionally interleave the local contour processing and the global contour processing in each processing node. When we use a coarse-grain machine such as SP-2, each processing node has a lot of data to be processed. The processing for the input data in each processing node can be divided into two categories. Processing of the local contour and part of the global contour having the starting pixel can be done independently of other processing nodes. However, processing of part of the global contour not having the starting pixel can only be done after performing the previous part of that global contour. A simple *priority-based* scheduling heuristic can contribute to improved performance. The contour pixels in each processing node are grouped into three priority classes: (1) *Priority 1*: global contours having their starting pixels in the processing node. (2) *Priority 2*: global contours having the local-starting pixels in the processing node. (3) *Priority 3*: local contours. Priority 1 is the highest priority class while Priority 3 is the

lowest priority class. By scheduling the tasks according to their priorities, we can reduce the idle time of a processing node. The goal of this task scheduling is to perform tasks on the critical-path first and keep the processing nodes. Details are shown in Figure 2.

In Step 1 (see Figure 2), we search the boundary of the 2-D contour pixel array in the problem and count the number of sends and receives. Then, we extract the starting and local-starting pixels from the 2-D contour pixel array and store them into a wait queue in Step 2. Step 3 receives all the pending tokens and updates the number of receives. Then, we check the wait queue. If any task has a token, it is extracted from the wait queue and inserted into the ready queue. If there is any global contour task in the ready queue, then we process it, otherwise, a local contour is processed. Step 4 takes a ready task for a global contour and performs the contour tracing and the approximation operations. If the task contains a local-ending pixel, then we send the token immediately and the number of sends is updated as necessary. This makes overlapping the next computation with the communication possible because each processing node may have independent tasks to perform. Step 5 takes a ready task for a local contour and performs the contour tracing and the approximation operations. Step 6 checks the number of sends and receives. If both of them are zeroes, then the processing node participate in the synchronization for termination.

B. Parallel Perceptual Grouping

Let *token* denote any of a segment, a line, a linear, a pair of parallel linears, a junction, or a rectangle. For example, a segment token contains information such as endpoint location, length, or slope of the segment. Let P denote the number of processors in the machine and let the image size be $n \times n$. Initially, we divide the image array into P blocks and distribute them to the processors for extraction of image features. Thus, each processor contains a subimage of size $\frac{n}{\sqrt{P}} \times \frac{n}{\sqrt{P}}$ along with image features detected during the feature extraction phase [7]. The image features detected are used as input to the perceptual grouping phase. We assume that the input to the grouping phase is a set of segment tokens. A segment token with starting coordinates (x_1, y_1) and ending coordinates (x_2, y_2) is stored in the processor which has the subimage containing (x_2,y_2) pixel. The *line* and *linear* tokens are also stored in this way before the grouping task begins.

B.1 – Outline of the Parallel Grouping Algorithm

We only analyze the Line Grouping task. The same approach can be applied to the Junction Grouping and other grouping tasks. Following the sequential algorithm described in Section 3.1, in the Line Grouping task, we group line segments which are closely bunched, overlapped, and parallel to each other to form a *line* (a linear structure at a higher granularity level). For each line segment, a search is performed within the region on both sides of the line segment within a constant width to find other line segments which are parallel to it. The detected segments are grouped to form a *line*.

The grouping algorithm is performed in a *scatter-and-gather* fashion. In the scatter phase, each processor computes the search window for each token stored locally and distributes *remote accesses* to perform search on subimage blocks overlapping with the search window. For example, if the search window of a segment in PE_2 overlaps with the subimage blocks

Procedure : Line-Grouping
Input: $n \times n$ image with line segment data embedded on the image at its two endpoint
Step 1: Distribute search requests for *long* segments
Step 2: Perform partial grouping operation
Step 3: Gather results
Step 4: Perform merge operations
Step 5: Perform barrier synchronization
end

Figure 3: A skeleton of parallel algorithm for Line Grouping.

contained in PE_3, PE_4, and PE_7, it will generate three search requests and send them to PE_3, PE_4, and PE_7 separately. In the gathering phase, partial grouping results are sent back to form new tokens. Initially, the segments are stored at the processor which has the subimage containing its ending pixel (x_2, y_2). To make complete image data available before performing window search, we also store the segment data at its starting point location (x_1, y_1) on the image. This indexing mechanism is also used in [13, 6]. The communication cost of performing the indexing step can be hidden because both endpoints are always located within the search window. The indexing can be done at the processor containing the starting point after receiving the search request.

For the sake of convenience, we define *short* segments as those segments whose search windows do not cross subimage boundary and *long* segments as those that cross at least one subimage boundary. For *short* segments, the grouping can be performed locally as the search window is within the subimage. The steps of our algorithm are outlined in Figure 3. In Step 1, for each *long* token, we compute its search window and send search requests to processors containing the subimage that overlaps with the token's search window. In Step 2, we first perform the indexing step and then perform search operations requested by the *remote* processors and those of local *short* segments. All those short segments complete this grouping task in Step 2. However, at this time, the search requests generated by *long* tokens results in partial grouping. These partial results need to be sent back and merged to form new tokens. In Steps 3 and 4, we send back the partial grouping results and generate new tokens upon receiving the partial grouping results. Note that the grouping process is decomposable and the partial result generated by each search request has $O(1)$ data size. The merge operation is a weighted-least mean square procedure to compute the average orientation of the new line. Only the partial sum of the angle, partial sum of the length of the grouped segments, and coordinates of the two farthest endpoints need to be sent back to the original processor to compute the line. Thus, Steps 3 and 4 have the same time complexity as Steps 1 and 2.

B.2 – Analyses of Communication Requirements

Each processor generates remote access requests for each *long* segments stored in it. All requests from a processor with the same destination can be packed into a single message.

Thus, each processor generates at most P messages of different lengths. Let l_{ij} be the length of the message to be sent from processor$_i$ to processor$_j$, $l_{max} = max(l_{ij}|0 \leq i,j \leq P-1)$. l_{max} is the longest outgoing message. Let $s = max(\sum_j l_{ij}|0 \leq i \leq P-1)$ and $r = max(\sum_i l_{ij}|0 \leq j \leq P-1)$ be the the maximum outgoing and incoming traffic generated at a processor respectively. Let $L = max(s,r)$. Based on our analysis we know $l_{max} \leq \frac{n}{2\sqrt{P}}$ and using the proposed algorithm in Section 3.3, we can control the communication traffic such that $L \leq \frac{2n}{\sqrt{P}}$. The data movement is a many-to-many personalized communication with bounded traffic L.

For image sizes used by the vision community ($n \leq 4K$) and machine sizes of practical message-passing machines ($P \leq 1024$), the scenario leads to $l_{max} \leq 512$ and $L \leq 2048$, for $n = 4K$ and $P = 16$. The total amount of data to be transmitted on the network, $R = \sum_{i,j} l_{ij}. 0 \leq i,j \leq P-1$, is at most $256K$ for $R = L \times P$, $n = 4K$ and $P = 1024$. This type of data communication is usually not a bulk data transfer but a sparse many-to-many communication having large message length variance. Based on the computational model used in this paper, a straightforward algorithm for many-to-many personalized communication requires $P(T_d + L \times \tau_d)$ communication time as the longest message size could be L at each communication step and P communication steps are performed. We refer to this algorithm as *1-stage* algorithm. This situation in the worst case may result in severe performance deterioration without careful design of an algorithm to smooth out the variation in the message lengths.

Some earlier studies have considered implementing data communication problems in distributed memory machines. In [14], a 2-stage algorithm for many-to-many personalized communication with message length variance has been studied. In the first stage, message i, for $0 \leq i < P$, is partitioned into P packets of equal size. They are designated as $packet_{i,0}, packet_{i,1}, ..packet_{i,P-1}$. The new ith outgoing message at each processor, for $0 \leq i < P$, is composed of $packet_{j,i}$, for $0 \leq j < P$. All P outgoing messages initiated from the same processor will have equal size after this step. In the first stage, P communication steps are performed and each outgoing message from processor i, has message length $\sum_{(0 \leq j < P)} \lceil \frac{l_{i,j}}{P} \rceil < \frac{L}{P} + P$, for $0 \leq i < P$. In the second stage, all processors receive P messages. Each outgoing message is composed of data having the same destination. Another P communication steps are performed to move the data to their destination. The algorithm takes $O(L + P^2)$ computation time and $2PT_d + 2(L + P^2)\tau_d$ communication time to perform the data movement. The techniques in [14] do not reduce the number of communication steps. If P is large, the startup time may dominate the total communication time.

Due to the relatively high startup time associated with a communication step, it is necessary to design communication algorithms which smooth out the message length variance and reduce the number of communication steps to avoid large startup time. We propose a 5-stage algorithm which can reduce the communication time for many-to-many data communication with high message length variance.

B.3 – A 5-Stage Algorithm for Communication

We propose a 5-stage algorithm which can reduce the communication time for many-to-many data communication with light traffic (small L) and high message length variance. We assume P is the number of processing elements (PEs) and each PE sends or receives at most L data. To describe the 5-stage algorithm, we use the following definitions and procedures.

We define two partitions; **Partition A**: $G_0, G_1, G_{\sqrt{P}-1}$, PE_i belongs to $G_{\lfloor \frac{i}{\sqrt{P}} \rfloor}$, for $0 \leq i < P$; **Partition B**: $G'_0, G'_1, G'_{\sqrt{P}-1}$, PE_i belongs to $G'_{i \ mod \ \sqrt{P}}$ for $0 \leq i < P$. The logical partitions of the processor array is used to distinguish the communication scheduling in each stage. Two procedures are used to prepare the outgoing messages. **Rearrange A**: this procedure deals with the incoming data in the first three stages. In each processor, the data with destination PE_i is collected to form the $\lfloor \frac{i}{\sqrt{P}} \rfloor$th outgoing message from that processor. **Rearrange B**: this procedure deals with the incoming messages in Stages 4 and 5. In each processor, the data with destination PE_i is collected to form the $(i \ mod \ \sqrt{P})$th outgoing message from that processor. Note that, there are always $\sqrt{P}$ outgoing messages prepared in each processor at all stages. In the following, we define three other main procedures:

1. $\sqrt{P}$**-decompose**: Assume each processor has $\sqrt{P}$ outgoing messages. Each outgoing message is partitioned into $\sqrt{P}$ packets of even size numbered $0, 1, ..\sqrt{P}-1$. These packets are then rearranged such that the new ith outgoing message is composed of the ith packet from all the $\sqrt{P}$ outgoing messages, for $0 \leq i < \sqrt{P}$.

2. $\sqrt{P}$**-shuffle**: We send the $\sqrt{P}-1$ messages to the other $\sqrt{P}-1$ processors within the same logically partitioned group based on **Partition A** or **B**.

3. $\sqrt{P}$**-collect**: PE_i collects the data destined for $\mathrm{G}_{\lfloor \frac{i}{\sqrt{P}} \rfloor}$ from all the other PEs within its logical group based on **Partition B**.

Lemma 1 *Given a message-passing machine having P processors, a many-to-many personalized communication with bounded traffic L can be performed in $O(L+P)$ computation time, $5\sqrt{P}T_d + 5 \times max(L,P)\tau_d$ communication time.*

Procedure: 5-stage Algorithm
Stage 1: Perform *Rearrange A*, $\sqrt{P}$*-decompose*, and $\sqrt{P}$*-shuffle* based on Partition **A**.
Stage 2: Perform *Rearrange A*, $\sqrt{P}$*-decompose*, and $\sqrt{P}$*-shuffle* based on Partition **B**.
Stage 3: Perform *Rearrange A* and $\sqrt{P}$*-collect*
Stage 4: Perform *Rearrange B*, $\sqrt{P}$*-decompose*, and $\sqrt{P}$*-shuffle* based on Partition **A**.
Stage 5: Perform *Rearrange B* and $\sqrt{P}$*-shuffle* based on Partition **A**.
end

Figure 4: A skeleton of the 5-stage algorithm for personalized communication.

The 5-stage algorithm is performed in three phases. The first phase is used to smooth the message length variance. In the first phase, two communication stages are performed

to redistribute the data such that data destined for G_i is evenly distributed among the P processors, for $0 \leq i < \sqrt{P}$. Thus, at the end of the first phase, each message at a processor has length at most $L \times \sqrt{P}/P = \frac{L}{\sqrt{P}}$. The second phase consists of a single stage. In this stage, we redistribute the data so that the data destined for processors in G_i are moved to processors in G_i, for $0 \leq i < \sqrt{P}$. During the third phase (Stage 4 and 5), the 2-stage algorithm is performed on the data within each group G_i, $0 \leq i < \sqrt{P}$, to move the data to their destination.

In each stage, each processor sends and receives $\sqrt{P}$ messages among processors within its own group. Assume the processors in each group are numbered by PE'_0, PE'_1,..$\mathrm{PE}'_{\sqrt{P}-1}$ following the order of their physical node number. Movement of the $\sqrt{P}$ messages in each stage can be realized by $\sqrt{P}$ rounds of data permutation within each logically partitioned group. A simple schedule can be used during each stage: in round j, $0 \leq j < \sqrt{P}$, PE'_i sends the $((i+j) \ mod \ \sqrt{P})$th message to $\mathrm{PE}'_{(i+j) \ mod \ \sqrt{P}}$ in parallel for all i and all $\sqrt{P}$ groups. $0 \leq i < \sqrt{P}$. Each stage takes $\sqrt{P}T_d + max(L, P)\tau_d$ time for communication, since during each round a processor sends at most $\frac{L}{\sqrt{P}}$ data. The 5-stage algorithm can be performed in $O(L + P)$ computation time and $5\sqrt{P}T_d + 5 \times max(L, P)\tau_d$ communication time.

B.4 – Time Complexity of the Grouping Algorithm

The number of *long* segments in a processor is bounded by $\frac{2n}{\sqrt{P}}$ as the detected line segments do not overlap and the number of such segments is less than half the number of the subimage boundary pixels. Let R be the total number of the search requests generated by the the *long* segments over the entire processor array. Each processor can receive at most $4 \times \lceil \frac{n}{2\sqrt{P}} \rceil$ search requests since the number of requests is bounded by the number of subimage boundary pixels. Thus, R is less than $P \times \frac{2n}{\sqrt{P}}$. Note that each processor may contain $\leq \frac{2n}{\sqrt{P}}$ long segments and each long segment may generate as many as $\sqrt{P}$ requests. Thus, the total number of requests generated in a processor can be as large as $\frac{2n}{\sqrt{P}} \times \sqrt{P} = 2n$. To smooth the variance of the outgoing traffic from the processors, we redistribute the long segments such that each processor generates $\frac{R}{P}$ requests and $\frac{R}{P} \leq \frac{2n}{\sqrt{P}}$.

Theorem 1 *Given an $n \times n$ image, the Line Grouping task can be performed in $O(n^2/P)$ computation time and $20\sqrt{P}T_d + 10(\log P)T_d + max(\frac{40n}{\sqrt{P}}, 20P)\tau_d$ communication time using P processors.*

We briefly outline an analysis of the computation and communication times in Theorem 1, following the steps shown in Figure 3. In Step 1, we first redistribute the long segments such that each processor generates $\frac{R}{P}$ requests. We then move these requests to their destinations using the 5-stage algorithm. Each processor computes the number of requests generated for each long segment and computes the sum of the number of requests (R) among the entire processor array. This can be computed by performing a sum operation to obtain R. Let $r = \frac{R}{P}$ be the average number of requests to be generated at a processor. To compute the destination of the long segments for redistribution, a prefix sum operation is performed on a length array with the ith entry storing the number of requests to be generated by the ith segment. The destination of the long segment can be specified as $\mathrm{PE}_{\lfloor \frac{len[i]}{r} \rfloor}$, where

$len[i]$ is the prefix sum value of the ith segment. The two cooperative operations (sum and prefix sum) take $4(\log P)T_d$ communication time. The 5-stage algorithm can be performed to move the long segments. Step 1 can be completed in $O(\frac{n}{\sqrt{P}} + P)$ computation time and $5\sqrt{P}T_d+4(\log P)T_d+max(\frac{10n}{\sqrt{P}}, 5P)\tau_d$ communication time. Step 2 can be performed in $O(\frac{n^2}{P})$ computation time to perform the search on a $\frac{n}{\sqrt{P}} \times \frac{n}{\sqrt{P}}$ subimage. During Steps 3 and 4, we gather the partial grouping results and perform the merge operation to form new tokens. Step 3 involves data communication. The communication can be completed in $O(\frac{n}{\sqrt{P}} + P)$ computation time and $5\sqrt{P}T_d + 4(\log P)T_d + max(\frac{10n}{\sqrt{P}}, 5P)\tau_d$ communication time. In Step 4, a similar procedure as in Step 1 can be performed to move back the results. Step 4 can be completed in $O(\frac{n}{\sqrt{P}} + P)$ computation time and $5\sqrt{P}T_d + 4(\log P)T_d + max(\frac{10n}{\sqrt{P}}, 5P)\tau_d$ communication time. Step 5 can be performed in $2(\log P)T_d$ communication time. The total execution to perform the Line Grouping task is $O(\frac{n^2}{P} + P)$ computation time and $20\sqrt{P}T_d + 10(\log P)T_d + max(\frac{40n}{\sqrt{P}}, 20P)\tau_d$ communication time. The computation time is $O(\frac{n^2}{P})$ for $n > P$; typically, $n \leq 4096$ and $P \leq 1024$. Due to space constraints, the analysis of time complexity of the Junction Grouping will be provided in the full version of the paper.

3.4 Parallel Implementation

In this section, we discuss the implementation details and performance results for parallelizing the selected IU tasks on IBM SP-2 and TMC CM-5.

A. Parallel Implementations of a Linear Feature Extraction Task on IBM SP-2

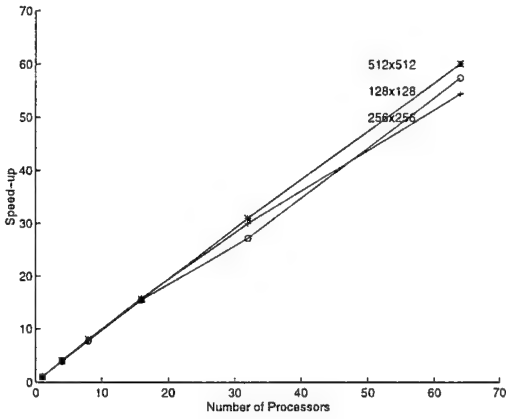

Figure 5: Speed-up of the Contour-Pixels Detection Phase

The algorithm was implemented on a SP-2 dedicated pool of 64 processing nodes at the Maui High Performance Computing Center. We used 1, 4, 8, 16, 32, and 64 processing nodes in the pool. The code was written using C and MPL message passing library. The total length of the code is around 3000 lines. The code has two parts: *manager* part and *worker* part. The manager part reads the image data to processing node 0 and then evenly distributes it to the processing nodes including processing node 0. The worker part is executed

by the processing nodes which perform the contour-pixels detection and linear approximation.

In the contour-pixels detection phase, we implemented the overlapping computation with communication technique. It performs the window operations for the boundary part of the tasks first, and sends the result immediately using the non-blocking command. Because we keep working on the remaining part of the tasks, we can hide the actual transfer time by overlapping next computation in the main processor with the communication performed by the communication processor. Furthermore, due to the relatively large non-blocking command execution time in SP-2, we employed the message packing technique used in [7, 10] by storing the individual messages for the same destination into an outgoing buffer and sending that buffer as a single message (similar technique is also used by compiler designers, see for example [5]). We show the speedup of the contour-pixels detection phase in Figure 5. Given an image of size 512×512, the contour-pixels detection can be performed in 0.05 seconds on a 64-node SP-2. A serial implementation takes 3 seconds on a single-node of SP-2. These reported data are calculated by the actual elapsed time using the wall clock in the dedicated-mode.

In the linear approximation phase, we also exploited the non-blocking command in the asynchronous implementation because the scheduling policy lets some tasks to be processed after the non-blocking command. In Step 3 of Figure 2, we used a non-blocking receive command. Once we post the commands, the main processor can start the next computational work while the actual receive operation is performed by a communication processor. Also, the main processor initiates the non-blocking send command in Step 4 of Figure 2. The next computational work can be overlapped with the actual send operation being performed by the communication processor. However, there is some difficulty when we apply the message packing technique in the asynchronous implementation. The communication pattern is irregular and is not known at compile time. If we determine the degree of the packing using the fixed-size outgoing buffer (each processing node sends the packed message when the outgoing buffer is full), then deadlock can occur; all processing nodes have no computational work to do and are waiting for the results of adjacent processing nodes, but no one can break this circular wait condition because all the outgoing buffers are not full. We avoid this deadlock scenario by sending the outgoing buffer after some fixed time. By doing this, each processing node can start next computational work with the received data.

The speed-ups of both synchronous and asynchronous algorithms are calculated with the elapsed time using the wall clock in the dedicated-mode. These are shown in Figure 6. The speed-ups of the synchronous algorithm become saturated soon due to idling of processing nodes caused by data dependencies and synchronization overheads. However, large speed-ups with the asynchronous algorithm is achieved by reducing the synchronization overheads and applying task scheduling. As the number of processing nodes increases for a given image data, the asynchronous algorithm gives large speed-ups compared with the synchronous algorithm. The asynchronous implementation on an image of size 512×512 completes in 0.015 seconds on a 64-node of SP-2. The execution time of the synchronous algorithm is 0.225 seconds on a 64-node of SP-2. A serial implementation of the linear approximation

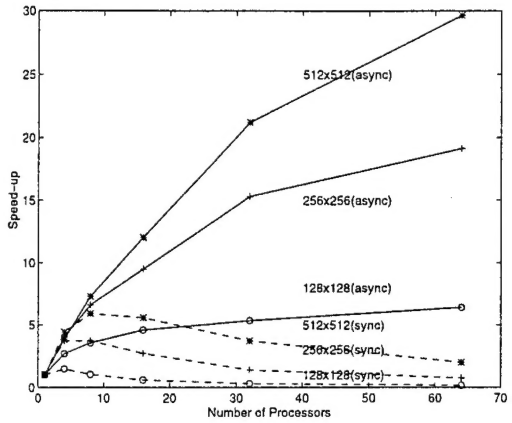

Figure 6: Speed-up of the Linear Approximation Phase

takes 0.445 seconds on a single-node of SP-2.

B. Parallel Implementations of A Line Grouping Step on TMC CM-5

We have performed perceptual grouping (the Line Grouping step developed by the vision group at USC) in about 0.324 *sec* using a CM-5 partition having 256 nodes (See Table 2). This task takes about 10 *seconds* on a state-of-the-art Spacrstation. The key problem in parallelizing perceptual grouping is unbalanced work load created by uneven distribution of features in the image. We have developed load balancing techniques that evenly distribute the load among the processors [7].

Total Execution Time (in *seconds*)			
Image size	Number of	CM-5 Partition Size (P)	
$(n \times n)$	Segments	P=64	P=256
AP1($n = 256$)	1594	0.215	0.108
Mall($n = 512$)	5474	0.622	0.243
J3($n = 1K$)	8943	1.014	0.324

Table 2: Execution times of the Line Grouping procedure on various partitions of CM-5.

4 Selected Publications

1. Yongwha Chung, Viktor K. Prasanna, and Cho-Li Wang, "A Fast Asynchronous Algorithm for Linear Feature Extraction on IBM SP-2," to appear at IEEE Workshop on Computer Architectures for Machine Perception, September 1995.

2. Cho-Chin Lin, Viktor Prasanna, and Yongwha Chung, "Data Remapping for Intermediate Level Analysis in Image Understanding on Distributed-Memory Machines," *Workshop on Solving Irregular Problems on Distributed Memory Machines, IPPS'95*, pp. 35-42, April 1995.

3. Viktor K. Prasanna and Cho-Li Wang, "Scalable Parallel Implementations of Perceptual Grouping on Connection Machine CM-5," in International Conference on Pattern Recognition, pp. 229-233, October 1994.

4. Viktor K. Prasanna and Cho-Li Wang, "Image Feature Extraction on Connection Machine CM-5," in Image Understanding Workshop, November 1994.

5. Cho-Li Wang, Viktor K. Prasanna, and Young Won Lim, "Parallelization of Perceptual Grouping on Message-Passing Machines," to appear at IEEE Workshop on Computer Architectures for Machine Perception, September 1995.

5 Professional Personnel

Ram Nevatia	Principal Investigator
Viktor K. Prasanna	Co-principal Investigator
Cho-Li Wang	Student Research Assistant
Cho-Chin Lin	Student Assistant

6 Interaction

We presented the paper titled "Data Remapping for Intermediate Level Analysis in Image Understanding on Distributed-Memory Machines" at the *Workshop on Solving Irregular Problems on Distributed Memory Machines, IPPS'95*, Santa Barbara, California, April 1995. The paper "Scalable Parallel Implementations of Perceptual Grouping on Connection Machine CM-5" was presented in International Conference on Pattern Recognition, Jerusalem, Israel, October 1994. We will present two papers, "A Fast Asynchronous Algorithm for Linear Feature Extraction on IBM SP-2" and "Parallelization of Perceptual Grouping on Message-Passing Machines", at the IEEE Workshop on Computer Architectures for Machine Perception in Como Italy, September 1995.

References

[1] J. Dunham, "Optimum Uniform Piecewise Linear Approximation of Planar Curves," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. 8, No. 1, pp. 67-75, 1986.

[2] IBM, *Parallel Programming Subroutine Reference Release 2.0*, 1994.

[3] C. Lin, V. Prasanna, and Y. Chung "Data Remapping for Intermediate Level Analysis in Image Understanding on Distributed-Memory Machines," *Workshop on Solving Irregular Problems on Distributed Memory Machines, IPPS'95*, pp. 35-42, April 1995.

[4] R. Nevatia and K. Babu, "Linear Feature Extraction and Description," *Computer Graphics and Image processing*, Vol. 13, pp. 257-269, 1980.

[5] D.J.Palermo et. al., "Communication Optimizations used in the Paradigm Compiler for Distributed Memory Multicomputers," Vol.II, *1994 International Conference on Parallel Processing*, II-1-10, 1994.

[6] V. Prasanna and C. Wang, "Scalable Parallel Implementations of Perceptual Grouping on Connection Machine CM-5," *International Conference on Pattern Recognition,* 1994.

[7] V. Prasanna and C. Wang, "Image Feature Extraction on Connection Machine CM-5," *Image Understanding Workshop*, pp. 595-602, 1994.

[8] J. Roberge, "A Data Reduction Algorithm for Planar Curves," *Computer Vision, Graphics, and Image Processing*, Vol. 29, pp. 168-195, 1985.

[9] C. Stunkel et. al., "The SP1 High-Performance Switch," *Proc. of Scalable High Performance Computing Conference*, pp. 150-157, 1994.

[10] C. Wang et. al., "Scalable Data Parallel Implementations of Object Recognition using Geometric Hashing," *Journal of Parallel and Distributed Computing*, pp. 96-109, March 1994.

[11] A. Huertas, C. Lin, and R. Nevatia, "Detection of Buildings from Monocular Views of Aerial Scenes using Perceptual Grouping and Shadows," *Image Understanding Workshop*, pp. 253-260, 1993.

[12] D. Lowe, *Perceptual Organization and Visual Recognition*, Kluwer Academic Press, MA, 1985.

[13] H. Lu and J. K. Aggarwal, "Applying Perceptual Organization to the Detection of Man-Made Objects in Non-Urban Scenes," *Pattern Recognition*, pp. 835-853, 1992.

[14] S. Ranka, R. Shankar, and K. Alsabti, "Many-to-Many Personalized Communication with Bounded Traffic," to appear in *The Symposium on the Frontiers of Massively Parallel Computation*, Feb. 1995.

[15] T. Kwan, B. Totty, and D. Reed, "Communication and Computation Performance of the CM-5," *Proc. of Supercomputing '93*, pp. 192-201, 1993.

[16] B. Lester, *The Art of Parallel Programming*, Prentice-Hall, Inc., 1993.